University of Ljubljana
Faculty of Computer and Information Science

Ožbolt Menegatti

Compiling Octave to Lua virtual machine

master's thesis

the 2nd cycle master's study programme
computer and information science

Supervisor: doc. dr. Boštjan Slivnik

Ljubljana, 2017

Copyright. The results of this master's thesis are the intellectual property of the author and the Faculty of Computer and Information Science, University of Ljubljana. For the publication or exploitation of the master's thesis results, a written consent of the author, the Faculty of Computer and Information Science, and the supervisor is necessary.

©2017 Ožbolt Menegatti

Acknowledgments

Zahvaljujem se doc. dr. Boštjanu Slivniku za pomoč in usmerjanje pri izdelavi magistrskega dela. Zahvala gre tudi vsem bližnjim za podporo med študijem in pisanjem magistrske naloge.

Ožbolt Menegatti, 2017

Whose motorcycle is this?
It's a chopper baby.
Whose chopper is this?
It's Zed's.
Who's Zed?
Zed's dead baby, Zed's dead.

— Bruce Willis as Butch Coolidge and Amanda Plummerin as Honey Bunny in Pulp Fiction

Contents

List of used acronmys

acronym	meaning
PEG	Parsing expression grammars
CFG	Context-free grammars
JIT	Just in time
GPL	GNU general public license
VM	Virtual machine
AST	Abstract syntax tree
IR	Intermediate representation
CPU	Central processing unit
(U)LEB 128	Unsigned Little Endian Base Encoding 128

Povzetek

Naslov: Prevajanje Octave v Luin navidezni stroj

Octave je višjenivojski programski jezik namenjen matematičnim problemom. Octave vsebuje interpreter, ki izvaja programe združljive s programskim paketom Matlab. V primerjavi z drugimi popularnimi interpreterji je Octave-ov počasnejši in s tem neprimeren za izvajanje časovno zahtevnih programov. To delo poskuša pohitriti izvajanje kode z uporabo navideznega stroja programa LuaJIT 2.0. Spisan je bil prevajalnik, ki prevede Octavine programe v Lua izvorno kodo ali LuaJIT bytecode. Povezava med Luo in Octaveom omogoča, da programi ohranjajo vse funkcionalnosti Octavea. Dosežene pohitritve so v intervalu med 1.1 in 6.3, odvisno od izvajanega Octave programa.

Ključne besede

prevajalnik, Octave, Lua

Abstract

Title: Compiling Octave to Lua virtual machine

Octave is high level programming language for numerical computations. It uses its own interpreter, to interpret Matlab compatible programs. The speed of programs, that are run with Octave's interpreter is slow in comparison to other popular interpreters and is not useful for implementing time consuming programs. This work attempts to speed up Octave programs using LuaJIT 2.0 interpreter. We have implemented a compiler, that compiles Octave scripts into a Lua source or LuaJIT 2.0 bytecode. Bindings were written between Octave and Lua to maintain most of the functionality of Octave. Speedups in range of 1.1 to 6.27 were achieved depending on the executed Octave program.

Keywords

compilation, Octave, Lua

Razširjeni povzetek

Matlab [1] je programski jezik in razvojno okolje, ki se uporablja za razvoj numerično zahtevnih programov. Zaradi svoje preprostosti je pogosto uporabljen v akademskem okolju in industriji. Octave [2] je odprtokodni nadomestek Matlaba, ki pa je počasnejši v izvajanju Matlab programov. To delo poskuša pohitriti Octave s prevajanjem teh programov v jezik navideznega stroja LuaJIT 2.0.

Obstaja več poskusov prevajanja Matlaba in Octavea v statično tipizirane jezike. Glavni problem v teh delih je določanje tipa spremenljivk in dimenzije matrik. Prevajanje Matlaba v C ali C++ v delih [3] [4] [5] vsebuje določanje tipov spremenljivk, da lahko program specializiramo za določen tip ob času prevajanja. Določanje tipov je v delu Paulsen et. al. [4] delno prepuščeno programerju, ki tipe določi ročno. Ko prevedemo Matlab kodo v C/C++ kodo, je potrebno matematične operacije podpreti s knjižico, ki jih implementira. Vsako delo uporabi različno knjižico za podporo matematičnim operacijam, od implementacije svoje knjižice v [3], pa do uporabe že obstoječič knjižic v [4] [5].

Jens Rucknagel [6] je napisal prevajalnik, ki prevaja iz Octavea v C++. Prav tako je v tem delu implementirano določanje tipa spremenljivk, vendar ob potencialnem neuspehu ne vrne napake. Namesto določitve statičnega tipa uporabi Octave knjižico, kjer so tipi spremenljivk dinamični.

Za ciljni jezik prevajalnika v tem delu je bil izbran programski jezik Lua. Zaradi svoje preprostosti, velikosti in hitrosti je pogosto uporabljen kot skriptni jezik v igrah. Preprostost jezika omogoča implementacijo hitrih interpre-

terjev z uporabo JIT prevajanja [7]. LuaJIT 2.0 [8] je interpreter, ki uporablja JIT in implementira verzijo 5.1 Lua jezika; ta implementacija Lue je tudi razlog, da je Lua ena izmed najhitrejših dinamičnih jezikov nasploh [9]. Dodatno argument za izbiro Lue kot ciljnega jezika je velikost LuaJIT 2.0 interpreterja, ki znaša 1 MB. Za primerjavo, Javin navidezni stroj OpenJDK zasede 37.91 MB. Ciljna jezika, ki jih to delo uporablja, sta Lua in LuaJIT 2.0 bytecode, ki je strojni jezik za LuaJIT 2.0 navidezni stroj. LuaJIT 2.0 lahko izvaja oba omenjena jezika.

V prvem delu prevajalnika je Octave program pretvorjen v abstraktno sintaksno drevo (ASD). Na prvem nivoju drevesa se nahajajo izjave, ki so lahko dveh vrst, blok ali izraz. Blok je sestavljen iz več izrazov, vsak blok pa ima tudi svojo vrsto, te so `for` zanka, `while` zanka, `do until` zanka, `if-elseif-else` veriga in deklaracija funkcije. Vsaka vrsta predstavlja eno izmed kontrolnih struktur v Octaveu. V Octave programu se vsak izmed naštetih blokov začne in konča s svojo ključno besedo, denimo `for` blok se konča s ključno besedo `endfor`. Poleg posameznemu bloku lastne končne besede, pa si med sabo delijo še skupno končno besedo `end`.

Druga vrsta izjave je izraz, izraz pa je zopet razdeljen v več vrst. Podizraz je oblika matematičnega zapisa z operatorji in vrednostmi. Vsak operator ima prioriteto, ki določa vrstni red operacij. Med podizraze spadajo tudi enostavno prirejanje (denimo `a = 1`) in sestavljeno prirejanje (denimo `a += 1`). Ukaz je posebna vrsta klica funkcije, ki je sestavljena iz zaporedja argumentov, ločenih z enim ali več presledki. Prvi argument je ime funkcije, ostali argumenti pa predstavljajo argumente te funkcije. Break ali continue izraza nadzorujeta potek izvajanja v zanki, globalna deklaracija pa predstavlja seznam spremenljivk iz globalnega obsega, ki so uporabljene v lokalnem obsegu.

Vrednost je osnovna predstavitev Octaveovih vrednosti. Definirana sta sestavljena in enostavna vrednost. Enostavni vrednosti so števila v šestnajstiškem (`0x1`), znanstvenem (`1.0e3`) ali desetiškem zapisu (`1.23`), nizi znakov (`"bostjan"`) in imena spremenljivke (`var`). Te se nahajajo v listih ASD. Poleg enostavnih

vrednosti imamo tudi sestavljene vrednosti, to so matrike (`[1, 2, 3]`) ter podizjave znotraj oklepajev (`'(0x1 + [1, 2])'`).

Da lahko podpremo več ciljnih jezikov smo definirali vmesno predstavitev. Ta predstavitev je poenostavljeno ASD drevo, ki vsebuje operacije, ki jih podpirata oba cilja jezika. Vmesna predstavitev je zopet sestavljena iz več tipov, ki ustrezajo tipom iz struktur v ASD. Tipi v vmesni predstavitvi so vmesne vrednosti, vmesne izjave in registri. Vmesne izjave so sestavljene iz vmesnih vrednosti. Vmesna vrednost predstavlja vrednost, ki bo zapisana v register v LuaJIT 2.0 bytecode ali v localni spremenljivki v Lua. Register pa predstavlja prostor, kamor lahko zapišemo vmesno vrednost. Definirane so tri vrste registrov, lokalni, globalni ali statični. Lokalni in globalni registri predstavljajo vrednosti v Octave programu iz lokalnega in globalnega obsega, statični registri pa so posebna vrsta lokalnih registrov, ki so rezervirani zaradi implementacijskih razlogov.

Pri pretvorbi iz ASD, poskrbimo za pravilno delovanje več funkcionalnosti Octavea, ki niso podprte v Lui. Prva sintaksna lastnost Octavea je prirejanje brez podpičja, katero povzroči izpis vrednosti, katera je bila prirejena. Druga lastnost je shranjevanje vrednosti izjavam brez prirejanja v spremenljivko `ans`. Zadnja podprta funkcionalnost pa je vračanje več spremenljivk iz funkcij v Octaveu. To rešimo z uporabo Lua tabel.

Prevajalnik je zmožen prevesti Octave program v dva ciljna jezika, Lua programska koda in LuaJIT 2.0 bytecode. Generiranje ciljnega Lua program iz vmesne kode je preprosto, saj se vsak element v vmesni predstavitvi preslika v odsek Lua programa. Implementacija drugega ciljnega jezika, LuaJIT 2.0 bytecode, je nekoliko težji. Razloga za to sta pomanjkanje dokumentacije in razhroščevalnih orodij.

Za predstavitev Octave spremenljivk med izvajanjem uporabljamo C++ razred `octave_value` iz Octaveove knjižice `libinterp`. Preko te knjižice so podprti tudi aritmetični, primerjalni ter drugi operatorji in klici vgrajenih funkcij. Zato moramo pri začetku izvajanja programa z uporabo LuaJIT 2.0 navideznega stroja inicializirati tudi Octave interpreter.

Prevajanje Octave v ciljni jezik in zagon Luinega navideznega stroja je napisano v jeziku Rust. Za potrebe inicializacije uporabljenih interpreterjev smo napisali povezavo od Octave interpreterja in LuaJIT 2.0 navideznega stroja do jezika Rust. Pri poganjanju kode LuaJIT 2.0 navidezni stroj uporablja Octave knjižico `liboctinterp` in za ta namen je bila razvita druga knjižnica.

Delo lahko razdelimo med delo opravljeno v LuaJIT 2.0 in delo, ki ga opravi Octave. Vzemimo primer vrstice Octave programa `a = b + 1`. Ta se prevede v Lua kodo, ki jo lahko predstavimo v naslednji obliki: `assign(a, sum(b, 1))`. Dekodiranje in zajem parametrov funkcij je v domeni LuaJIT 2.0. Določanje tipa spremenljivke `b` in izvedba funkcije `sum` je v domeni Octavea. Zadnji korak, zapis vrednosti v spremenljivko, je v domeni Lue. V primeru zank in pogojnih stavkov je določanje pogojnih vrednosti v domeni Octave. Funkcionalnost zank, pogojnih stavkov in definicije ter klici funkcij so podprte z Lua strukturami, le določanje vrednosti pogojnih stavkov opravlja Octave.

Za potrebe merjenja hitrosti so bili pripravljeni štirje programi. Ti programi so implementacije naslednjih algoritmov: urejanje z mehurčki, množenje matrik, rekurzivno računanje Fibonnacijevih števil ter reševanja problema osmih kraljic. Zadnji program je spisan dvakrat. verzija v1 predstavlja direktno metodo, verzija v2 pa vsebuje dve pohitritvi, ki skrajšata čas izvajanja.

Izkaže se, da največje pohitritve dobimo pri močno rekurzivnih problemih (Fibonacci), manjše pri iterativnih, še manjše pa pri problemih, kjer se večino časa program nahaja znotraj vgrajenih Octave funkcij (osem kraljic). Pohitritev je predstavljena v tabeli 1.

$$S = \frac{t_{old}}{t_{new}} \tag{1}$$

Z uporabo navideznega stroja LuaJIT 2.0 smo uspeli doseči krajše čase izvajanja v primerjavi z Octave interpreterjem. Pohitritve (po formuli 5.3) so med 1,1 in 6,27. Manjše pohitritve opazimo v programih, kjer večino časa

Tabela 1: Pohitritev (po formuli 1) izvajanja pri različnih alogitmih

Ciljni jezik	`Lua`	`LuaJIT 2.0 bytecode`
`Mehurčno urejanje`	$1,89$	$1,89$
`Matrični produkt`	$2,17$	$2,20$
`Fibonacci`	$6,27$	$6,17$
`Osem kraljic, v1`	$1,16$	$1,15$
`Osem kraljic, v2`	$1,41$	$1,40$

izvajanje poteka v vgrajenih Octave funkcijah, boljše pohitritve pa dosežemo v močno rekurzivnih in iterativnih problemih.

Chapter 1

Introduction to Octave and MATLAB

Octave is an interactive programming language intended for numerical calculations. It was originally developed as a clone of MATLAB tprogramming language and released under the GPL license. MATLAB's core functionality of running MATLAB programs is mostly implemented except few minor differences (more in section 1.1). The language generalizes operations on scalars to vectors and matrices making it well suited for vectorized numerical calculations. Many programmers consider it easier to prototype algorithms using Octave/MATLAB instead of conventional languages such as C++. There are mutliple reasons for this:

- interactive development environment with facilities for debugging and analysis (see figure 1.1),
- dimensions and types of arrays are not necessary to specify in advance and
- large number of functions and higher level operators for matrix manipulation.

Octave language is syntactically similar to Ruby or Python. For example,

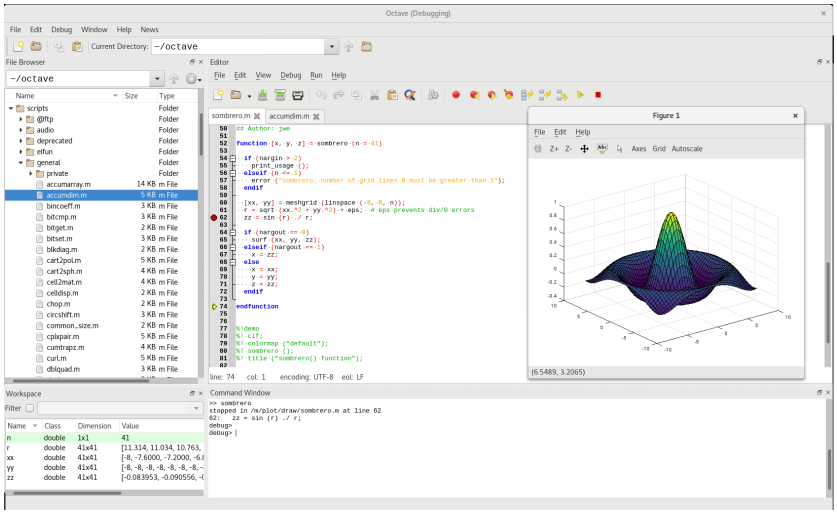


Figure 1.1: Octave graphical user interface

a scalar vector multiplication in pure Ruby would need a loop that iterates over both vectors, as in listing 1.1.

Listing 1.1: Scalar vector multiplication in Ruby

```
1 def multiply(a, b)
2     out = 0
3     for i in 0..(a.length - 1) do
4         out += a[i] * b[i]
5     end
6     return out
7 end
```

In Octave, a translation of this program does not change much:

Listing 1.2: Scalar vector multiplication in Octave

```
1 function out=multiply(a, b)
2     out = 0;
3     for i = 1:length(a)
4         out += a(i) * b(i);
5     end
6 end
```

However in Octave the same functionality is usually implemented as a

multiplication between two vectors: `out = a * b`, making the code much more concise. To see more examples of Octave programs, see listings A.1, A.2, A.3 and A.4.

The use of Octave can be found in the fields of psychology [10], neuroinformatics [11], electrical engineering [12] and many others. Many libraries that are written in other languages (for example a brain computer interface research library [13] and an electrical machine design library [14]) can also be accessed from Octave programs, which widens the set of available functionality of Octave.

1.1 Comparison between Octave and MATLAB

Octave was designed to run MATLAB programs; every MATLAB program, can also be executed using Octave's interpreter. Any differences in the core language semantics are considered a bug. However, MATLAB does provide many functions that are missing in Octave, for example image and signal processing, fuzzy logic, instrument control, and statistics functions. Many of these functionalities are provided to Octave using external packages and shared on Octave-forge website.

In addition to these missing core functions, MATLAB also distributes a list of GUI toolboxes, MATLAB-to-C compiler, Simulink modeling environment and other tools. Newer releases of MATLAB can also provide new features that are not distributed with Octave.

Octave on the other hand provides additional unique features. They range from syntactic additions to the language (such as `#` comments and line continuation using backslash) to test functions, documentations strings, `++` and `--` operators, and others. For the full list, see the Octave's wiki page [2].

Comparisons of the speed of Octave and MATLAB interpreters can be found in the work by Aurélien Garivier [15]. In this work we compared multiple interpreters of programming languages that are used for numerical

Programing language	`Execution time in seconds`
`Python`	296
`R`	333
`Scilab`	145
`MATLAB`	76.5
`Octave`	691
`C`	0.94

Table 1.1: The Baum-Welch algorithm for hidden Markov Models: speed comparison between Octave, Python, R, scilab, MATLAB and C from [15]

calculations. The results of comparing the speed of executing semantically the same programs in different languages are shown on table 1.1. It shows that MATLAB is an order of magnitude faster than Octave, while other higher level languages score in between the two.

1.2 Octave implementation

The Octave project contains two libraries, interpreter library called *liboct-interp* and library of basic data structures called *liboctave*. Both of these libraries are written in programming language C++. It is possible to use these basic data structures and write fast code using *oct programs*. These are C++ source files that can implement an Octave function. That function can either be called from an Octave's interpreter, or can be build as a standalone application. *Oct programs* are analogous to source MEX files for MATLAB.

Octave's *liboctave* library uses different GPL compatible libraries to perform computation on matrices. These are blas and lapack for linear algebra problems, arpack for computing large eigen value problems, fftw for computing discrete Fourier transform and others.

Octave's `libinterp` implements an interpreter that converts the input

source code to an abstract syntax tree (AST), where each node denotes a construct occurring in the source code. Once the AST is obtained, the interpreter recursively visits each node and performs an action that is defined for that node type. For example, a node in AST can represent an addition. That node would have a left and a right child node, each representing the left and the right side of an addition. Actions that would be performed, would be obtaining the values for the left and the right child and then the sum of these two numbers would be returned. This type of interpreter is called a tree-walk interpreter.

1.3 Related work

There were multiple attempts to compile MATLAB and Octave to C, C++ or Fortran. The main challenge with compiling Octave/MATLAB to statically typed language is determining types of variables. Type inference is a method of determining the types of variables at compile type using automatic deduction of data types of an expression. Though a dynamically typed language is not designed to contain type information, a limited form of compile time type inference can be performed. This type information enables additional optimizations.

One such attempt to compile Octave to C++ was presented in PhD thesis by Jens Rucknagel [6]. It uses partial type inference and compiled C++ code is on average 38 times faster than Octave's interpreter.

There were multiple attempts at compiling MATLAB to lower level languages [3] [4] [5].

Paulsen et. all. [4] implemented a MATLAB to C++ compiler for the purposes of compiling SeismicLab MATLAB library to C++. They use Antlr parser generator and Armadillo, a C++ linear algebra library. Although a simple type inference is implemented via single AST walk, a separate Python file is recommended for type and shape information.

Joisha and Banerjee [5] also worked on MATLAB to C compiler. Sim-

ilarly to [6], this work also centers around the shape and type inference at compile time. It uses static single assignment form [16] for its intermediate representation. This work also implements a much better compatibility with MATLAB, copying its behaviour in functions like sqrt, absolute value, NaN and Inf propagation, in cases where it is not compatible with the IEEE 754 standard [17].

In an older work from 1996, a MATLAB to Fotran 90 compiler called FALCON [18] was developed by De Rose *et al.* In addition to type and shape inference, this compiler also tried to perform other performance improvements. Algebraic restructuring of the code used algebraic rules defined for the variables to perform symbolic computations, which can simplify the code. For example, an expression $a = 2 * b + b$ is simplified to $a = b$. Parallelization module was also implemented and was used for running known side effect free MATLAB functions in parallel.

George Almasi and David Padua [19] implemented an interpreter for MATLAB that uses just in time and ahead of time compilation. It is a continuation of work done on FALCON compiler and uses the same techniques for code optimization. Interpreted code is up to two orders of magnitude faster than MATLAB interpreter, but still falls behind the speed of statically compiled MATLAB code using FALCON compiler.

Chapter 2

Using LuaJIT 2.0 to make interpretation faster

As this work is about making Octave programs running faster using Lua technology, this chapter introduces virtual machines (VMs), just in time compilation (JIT) and their use within Lua ecosystem.

2.1 Virtual machines and JITs

Virtual Machine is a software implementation of either existing or a fictional hardware platform. There exist two kinds of virtual machines. System virtual machine provides a substitute for a real machine and provides a functionality to execute an entire operating system. A process (or application) virtual machine on the other hand is designed to execute computer programs in a platform-independent environment. The term virtual machine (VM) in later sections refers to a process virtual machine.

Virtual machines were popularized by Pascal-P system [20], which included a compiler that produced code for a virtual stack machine. This code was named p-code and the term was later generalized to any code executed by a virtual machine. The term bytecode, used in conjunction with several modern programming languages, is synonymous with. For executing the

p-code, the VM can use:

- interpretation (for example the aforementioned Pascal-P system),
- just in time compilation to machine code (for example Microsoft Common Language Runtime [21]), or
- ahead of time compilation (for example Android Run Time before Android 7.0 [22]).

Most virtual machines however use a combination of two or more described methods.

Just in time compilation (JIT) is compilation that takes place during the execution of the program rather than before it. The target may be machine code (as is the case with all JIT compilers discussed in this work) or another target language.

The term just in time compilation is generally attributed to work on LISP by John MnCarthy in 1960 [23].

First widely used implementations were in languages Smalltalk and Self. For those implementations Mitchel *et al* [24] observed that compiled code can be obtained by storing actions performed during interpretation.

Challenge when implementing a JIT compiler is to decide when and how the code should be optimized. If a code that will be executed infrequently is optimized, then optimization may take longer than time savings obtained at with the optimization. To minimize the execution of a program, we must also take into account the time needed for its compilation and optimization.

Identifying frequently executed code segments or hot spots can be done in several ways. Java HotSpot VM before 2012 [25] monitors the program execution and identifies methods, where the program spends most of its time. Those spots are then compiled into optimized machine code. This approach is named method-based JIT.

A different approach to identification of hot spots is called tracing JIT. With this approach, intermediate representation contains special opcodes

that are used as counters. These instructions are placed at the beginning of every loop and every function. If one of these instructions is executed multiple times, then the code segment for that loop or function is compiled and optimized. LuaJIT 2.0 and modern versions of HotSpot VM [26] also implement this type of JIT.

With modern CPUs a JIT compiler can compile code segments in parallel with interpretation, which means that while a code segment is being interpreted on one core, JIT compiler can compile the same segment on an another core. If CPU has enough cores and these cores are not busy with other tasks then the compilation does not affect the speed of interpretation and the compiled version can be used when compilation is complete.

A JIT compiler can use the same optimizations that are found in ahead of time compilers. However, a JIT compiler can make use of additional information that can only be obtained at run time and use this information to further optimize the code. One such information is the number of times a function is used or a loop is iterated over. If a code segment is particularly hot and it has already been compiled and optimized, it can be compiled again, with stronger optimizations [27] [28]. The compiler can also use live profiling data to perform additional optimizations that improve the speed of produced code. These optimizations are type specialization, loop and recursive unrolling, instruction rescheduling and others.

2.2 Lua and LuaJIT 2.0

Lua [29] is a small, embeddable, dynamically typed programming language originally designed in 1993. It provides a small set of general features that can be used to implement different programming paradigms. Lua implements a single native data structure, table, that can be used as a map, array or set. Metatable is a table that allows us to change the behavior of a table. For instance, using metatables, we can define how Lua computes the expression a+b, where a and b are tables. Metatables also enable programmer to write

Language implementation	Size
Pypy	95.11MB
NodeJs	20.04MB
OpenJDK	37.91MB
LuaJIT 2.0	1.0MB

Table 2.1: Size comparison of different language implementations on Linux PC

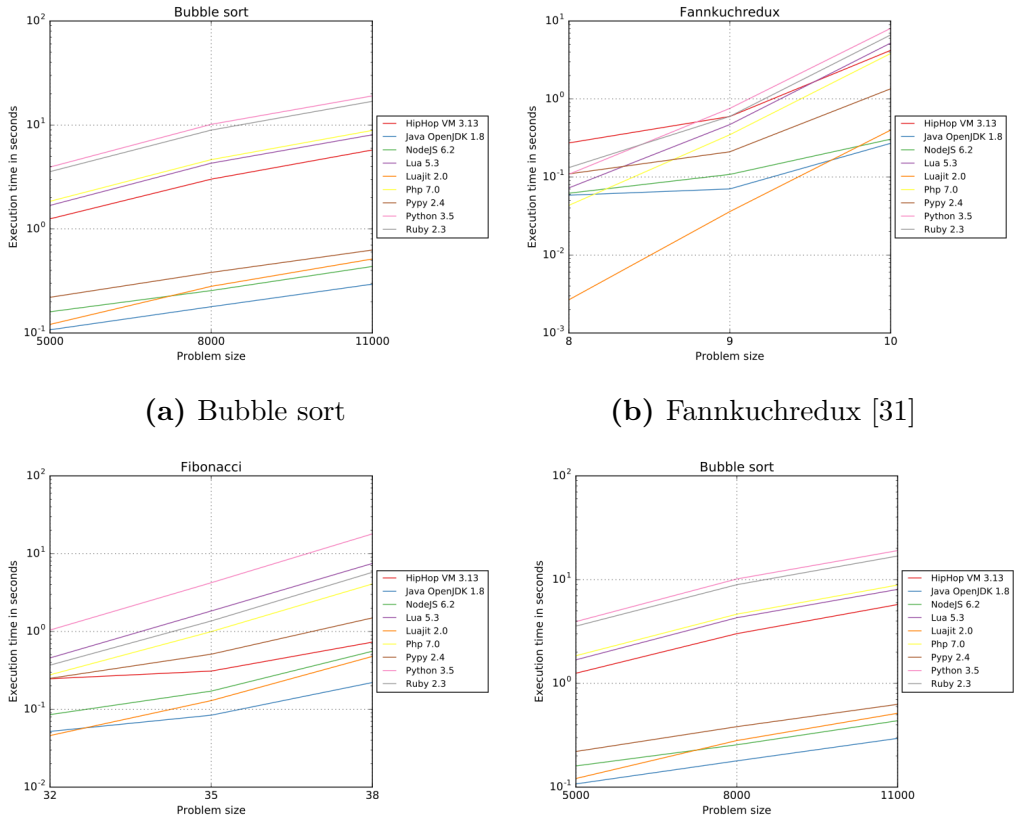
code similar to object oriented languages.

Lua does provides first class functions, enabling programmer to use many techniques from functional programming. Language has automatic memory management with incremental garbage collection. The implementation by original Lua authors uses a VM that interprets the Lua bytecode. Packaged size of the last language release 5.3 of Lua to Lua bytecode compiler, Lua interpreter and the standard library is only `227kB` (Linux, amd64 architecture).

We selected Lua as the target compilation for Octave because of its execution speed an small VM size. To benchmark different possible target languages we took multiple programs from *The Computer Language Benchmarks Game* [30]. We wrote a script that runs semantically identical programs implemented in different languages [9]. Results can be seen on figures 2.1a, 2.1b, 2.1c and 2.1d. The fastest language was Java, second fastest was LuaJIT 2.0, followed by NodeJs distribution of Google's V8 javascript engine (uses interpretation and JIT compilation) and Python's interpreter Pypy. The sizes of these four VMs, as listed on Pacman package manager on Arch Linux, architecture amd64 can be seen on table 2.1.

LuaJIT

Authors of LuaJIT Mike Pall and author of JavaScript Brendan Eich discuss Lua's design choices that lend themself well for JIT and other speed



(a) Bubble sort

(b) Fannkuchredux [31]

(c) Fibonacci

(d) N-body gravitational simulation

Figure 2.1: Plots of execution speed for multiple algorithms implemented in different programming languages and interpreters.

optimizations [7].

LuaJIT 2.0 [8] is an attempt to speed up Lua VM using JIT. LuaJIT 2.0 implements version 5.1 of Lua language and in addition provides efficient foreign function interface to C through a library called `ffi`.

Bytecode for LuaJIT 2.0 is defined in two source headers in LuaJIT 2.0 source code:

- `lj_bcdump.h` defines a bytecode file, headers, prototypes etc.,
- `lj_bc.h` defines bytecode instruction formats and lists all possible instructions.

Additional documentation is provided through the official LuaJIT 2.0 documentation on project's website [8]. The bytecode is defined per release and will change with newer versions of LuaJIT. In this work, we use LuaJIT version 2.0.

LuaJIT 2.0 bytecode can be written to a binary file with extension *luax* and then executed the same way as an ordinary Lua file. The file is constructed from multiple prototypes, where each prototype represents a function. The last function is the main function, which contains everything in the root scope in the Lua source file. Each prototype is composed from from five parts:

1. header, containing the prototype length, number of function parameters, framesize (number of registers used in a prototye) and information about the length of the following parts,

2. bytecode instructions,

3. kgc (garbage collected constants) list, which contains strings, local functions and table skeletons,

4. knum (number constants) list and

5. upvalue list, which consists of variables that a function captures from outer scope.

Variable size numbers that are used in a header, kgc and knum list are encoded using ULEB128 (unsigned little endian base 128 code) encoding.

To encode an unsigned number using ULEB128, its binary representation is first padded to a multiple of seven bits and then split into groups of seven bits. The groups are output from the least significant to the most significant group. The data is stored in the lower 7 bits, where the most significant bit of each byte is set to one except to the last byte, where it is set to zero [32].

Format	Byte 0	Byte 1	Byte 2	Byte 3
ABC	B	C	A	OP
AD	D		A	OP

Figure 2.2: ABC and AD bytecode instruction format in LuaJIT 2.0 bytecode.

Bytecode instructions

Bytecode instructions is a list of instructions that are 32 bits long and can be formatted in two formats, ABC or AD, as presented on table 2.2. OP slot in an instruction denotes the operation and A, B, C and D slots are the arguments for the operation. There exist multiple types of arguments:

- register number - there are 256 8-bit registers in LuaJIT 2.0 that are used for storing local variables and temporary values,
- literal - for loading numeric value into a register,
- lookup value - for loading values from kgc, knum or upvalue list,
- jump offsets and
- others, which are not used.

An example of instruction is presented below. This addition sums registers one and two into register three. An operation for this instruction is ADDVN. The description of this instruction is:

OP	A	B	C
ADDVN	dst	var	num

In the slot OP is an addition operation `ADDVN`, which a register `var` and an 8-bit number `num` into register `dst`. These are located in slots B, C and A respectively.

Knum list

In knum list is a list of number constants that are encoded using ULEB128. Each number can be either one of two types, a floating point number or an integer. A type of a number is determined with the least significant bit, using these formulas:

$$KNUM_{int} = x * 2 \tag{2.1}$$

$$KNUM_{float} = x * 2 + 1 \tag{2.2}$$

For example, we can take a knum list:

10000010100111011111100111110

00010001010010001001110110000

11110111110011110000101

Every element is ULEB128 encoded, so we need to divide this bit stream into 8-bit chunks.

10000010

10011101

11110011

11100001

00010100

10001001

11011000

11110111

11001111

00000101

This can be divided into two elements. The last 8-bits of these two elements start with 0, which means that the first number is constructed from first five bytes, and the second from the last five bytes. Next, remove most significant bits, to get chunks of seven bits.

$$00000100011101111001111000010010100$$
$$00010011011000111011110011110000101$$

The first number is odd and the second is even. Using the formulas 2.1 and 2.2 we can see that the first number is an integer and the second is a float. To get the real values, the least significant bit must be removed.

$$100011101111001111000010010100 = 299792458$$
$$10011011000111011110011110000101 = 2.99792458 \cdot 10^8$$

Kgc list

Kgc list is a list of garbage collected constants, where each element is constructed from a tag and the data. A tag is ULEB128 encoded number that represents a type of data. There are multiple types that are used in LuaJIT, but this compiler only uses the string type. The tag for this type is $8 + len(str)$. This tag does not collide with other tags, since their values are all smaller then eight. The data for string type is ASCII encoded string. An example string `Janez` would be tagged with tag 13 and encoded in kgc list as such:

$$00001011 \quad 01001010 \quad 01100001 \quad 01101110 \quad 01100101 \quad 01111010$$

Chapter 3

Compiler and the runtime system

The core of this work is the implementation of compiler from Octave to Lua virtual machine and development of the corresponding runtime environment in which the program is run. The compilation is divided into three steps:

- parsing Octave source code into abstract syntax tree (AST),
- converting AST into internal representation and
- generating Lua code.

The generated code uses structures from Octave's `libinterp` library and is run by LuaJIT 2.0 VM. All this is orchestrated by a Rust program, which sets Lua variables loads a library that connects LuaJIT 2.0 VM and `libinterp`. More on this runtime system in section 3.4.

3.1 Parser and the AST

Parsing Octave programs is performed using Rust library called rust-peg [33]. The library itself is implemented as a syntax extension for the Rust programming language, which takes a user defined grammar and generates a parser that is based on Parsing Expression Grammars [34].

3.1.1 Block

Octave source is parsed into a tree, where all children nodes of root node are statements. Every statement is separated with a newline or a semicolon. A statement can either be an expression or a block. Five different kinds of blocks are supported:

- `for` loop,
- `while` loop,
- `do until` loop,
- `if elseif else` chain and
- function declaration.

Every listed block in Octave program begins and ends with its own keyword. A common block end keyword is `end`, but every block also has its own end keyword; these are `endfor`, `endwhile`, `endif` and `endfunction`. Every block has children of type statement, but also a special child containing information about this block. This special child is different for each block type (also called `BlockKind`).

In actual Rust implementation of aforementioned structures, a block is a structure containing a list of statements and a BlockKind type. This implementation can be viewed on listing 3.1.

Listing 3.1: Block enum in Rust

```
1 enum BlockKind {
2     For(Operation),
3     While(SubExpression),
4     DoUntil(SubExpression),
5     IfElse(IfElse),
6     Func(Function),
7 }
8 struct Block {
9     content: Vec<Statement>,
```

```
10      kind: BlockKind,
11  }
```

In the next sections, when a structure is presented as "a choice of", as is represented BlockKind, in Rust that type is presented using enum. Enums in Rust are additive types that are implemented as tagged unions.

3.1.2 Expression

An expression can be one of multiple kinds:

- assignment,
- subexpression,
- command,
- break or continue keyword,
- global variable declaration.

An expression also has an additional property, a trailing semicolon. If semicolon is not present and an expression returns a value, than that value is displayed.

3.1.3 Assignment

An assignment in Octave can be a simple assignment, as known in most programming languages (eg. `a = 3`), or a more complex assignment. An example can be found on listing 3.2.

Listing 3.2: example complex assignment in Octave

```
1      A(3) += 4;
```

This complexity of assignment means that every assignment must be constructed from three parts:

- an identifier,
- zero or more expressions inside brackets on the left side,
- an expression on the right side and
- one of fourteen assignment operators in between.

3.1.4 Subexpression

An Octave subexpression tries to imitate classical mathematical expression with multiple variables and operators. Every operator has a priority, similar to how multiplication has priority over addition in real numbers. Parsing these structures is done using a list of parsing rules, where we first try to match an operator with highest priority and then we descent to operators with lower priority. An example from the grammar can be seen on listing 3.3.

Listing 3.3: Grammar for parsing some binary subexpressions

```
1 subexpression -> Subexpression = #infix<value> {
2         #L v1 "+" y2
3            v1 "-" y2
4         #L v1 "*" y2
5            v1 "/" y2
6         #R v1 "**" v2
7            v1 "^" v2
8         ...
9 }
```

Most subexpressions contain left and right side with operator in between. This kind of notation is called infix notation. Operators group left to right; this is implemented using precedence climbing [35], which is implemented as part of rust-peg library.

Octave contains three subexpressions that are formatted differently. These are ranges, function calls and prefix operators. Ranges use ':' as an operator and can be constructed in three different forms:

- left and right side (a:b),
- left, right and middle part (a:b:c),
- without any parameters (:).

Function call is a syntax for calling functions and indexing matrices. Let's take an example in listing 3.4.

Listing 3.4: Example function call in Octave

```
1 foo(bar, 3)
```

Postfix operators is only one type of operators that have higher priority than functions. There exists two such operators, increment ++ and decrement –.

3.1.5 Values

At the end of subexpression rules are actual Octave values, which are located at the leafs of the AST. Values can be divided into atoms and constructed values. Atoms are hexadecimal number (`0x123`), identificator (`var`), exponential number (`3e10`), integer (`123`), float (`123.123`) and string (`"bostjan"`).

Constructed values are constructed from subexpressions. There are two kinds, matrix (`[1, 0x2, 3]`) and subexpression within brackets (`(1 + 2e2)`).

In order to parse values, we need to recursively parse sebexpressions. However to parse subexpressions, we need to parse atoms. This recursive definition could in specific cases cause stack overflow. However since Rust uses LLVM segmented stack support to check for stack overflows, the parser is safe in terms of memory safety [36]. In development, stack overflow was never reached.

3.1.6 Command

Command is a special kind of function call. It is a list of strings, where first is interpreted as the name of the function and the rest of the list presents its

parameters. Parameter types are strings. These two lines are equivalent:

```
1 format long
2 format("long")
```

First line represents a command and a second line represents a function call. Also, these two lines are equivalent:

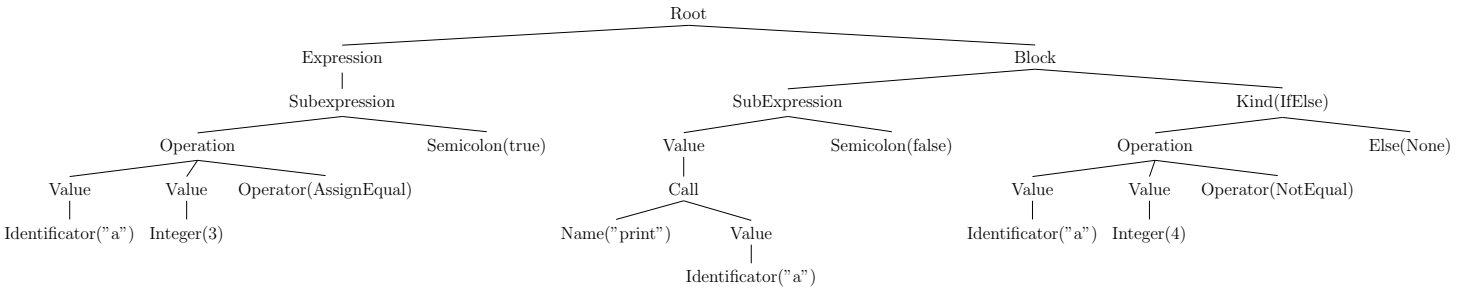
```
1 true
2 true()
```

We can see that invoking a command without parameters is equivalent to a variable in Octave. This ambiguity is described in section 3.2.4.

3.1.7 Example

Octave program on listing 3.5 is converted to an AST that is visually presented on figure 3.1.

Figure 3.1: An example abstract syntax tree



Listing 3.5: Example Octave source converted to AST

```
1 a = 3;
2 if a != 4
3     print(a)
4 end
```

3.2 Intermediate representation

Abstract syntax tree that is obtained from the parser is not directly translatable to Lua code, which is the reason an intermediate representation is used. The IR is a simplification of AST into a form that can be then used to generate the target code. It is constructed from three structures: register, intermediate value and intermediate statement. Names for these structures are similar to names of some structures in AST and they are related, but are not the same structures.

3.2.1 Register

A register represents any Octave value that will be assigned to a register int LuaJIT VM. There are three kinds of registers: local, global or static.

Local register presents a local variable and because of LuaJIT 2.0 limitations, 208 local registers are allowed. LuaJIT 2.0 itself provides 256 registers. Additional temporary registers are used for function calls, global registers and other operations. We chose 208 as a limit because of easier debugging, since 208 is presented as 0xD0. When debugging, every register over 0xCF is known to be temporarily created. Having a limited number of registers means that they need to be freed after they are not in use anymore. We have been able to minimize the use of registers enough that every tested program successfully compiled. If the limit is reached, than the compilation exits with an error.

Global registers are represented with a string and present global variables in Octave programs.

Static registers are special registers, for which we prealocate registers in LuaJIT 2.0. Two kinds exist, first presenting the table of local functions within one Octave source file. The second static register is the `ans` registers, which is used every time the result of an expression in Octave is not assigned to any variable.

3.2.2 Intermediate Value

Intermediate Value present a value that can be assigned to a register. There are seven kinds, first three are string, float and integers are literals from Octave programs. Fourth kind is a function call that is a structure which consists of a name, arguments, number of returned elements and a kind of call. There are three different kinds of function calls.

- Octave call, where we call a function that is is distributed with Octave as one of its integrated functions.
- Second type is Lua call, where we call any locally defined functions inside Octave program.
- The third kind is called `Oml` and presents any call to functions that are defined in a Lua library, which binds the generated code to Octave. This bindings are presented in section 3.4.

Table atoms

Last two kinds of intermediate values are nth element and a table. They represent the creation of Lua table and accessing its elements. Lua table is used for arguments when calling locally defined functions and obtaining the results from these functions.

3.2.3 Intermediate statement

An intermediate statement corresponds to a statement in AST. This structure has seven kinds, assignment, if else chain, for, while and function block and continue and break statement. These are fairly direct conversions from its equivalents in AST, the only conversion is combining `while` is and `do until` loops into unified representation.

3.2.4 Conversion

While much of the conversion from AST to IR is fairly direct, some isn't. There is exist functionality in Octave language that has to be expanded to multiple actions in Lua code. These are

- Assignment without trailing semicolon causes the variable to be printed to screen. To provide this feature, an additional call is made that prints the assigned value after the assignment.
- Expression without assignment is automatically assigned to a hidden variable ans. This is implemented with an additional assignment.
- Functions in Octave can return multiple values. In generated code this is solved by returning a table and performing a sequence of assignments that unpack the table.

Next semantic behaviour, that needs to be handled manually is runtime dependent behaviour. This is behaviour of an Octave program that is not defined at compile time, but rather at runtime. One approach of solving this problem is generating the code that handles these cases at runtime, which results in slower code. An alternative is to deviate from semantics of Octave and in such cases define our own behaviour. In this work we avoided these cases by not supporting certain Octave functionalities (eg. anonymous functions), which makes the generated code compatible with Octave without the need for handling different cases at runtime.

Some Octave code can contain context dependent behaviour, which means that the same line of code can perform different actions depending on the previous lines of the same program. In Octave programs we found two cases of context dependent behaviour, which are command versus variable ambiguity and function call versus matrix indexing ambiguity. can be found on listing 3.6.

Listing 3.6: pi as a command and as a variable

```
1 a = pi;      # a <- 3.141592653589793
2 pi = 3.14;   # pi <- 3.14
3 a = pi + 1; # a <- 4.14
```

On the first line, we are using a variable name `pi` that is not yet defined. We originally thought that `pi` is a predefined variable in Octave whose value is a computer estimation of constant π. Instead, `pi` is actually a command that calls the function named `pi`, which returns the value of π. On the second line, `pi` is assigned a value of exactly `3.14`. Now on the third line, pi is no longer a command, it is a variable that is defined on the second line.

Example of function call versus matrix indexing ambiguity see listing 3.7

Listing 3.7: exp as a function call and as matrix indexing

```
1 a = exp(1)     # a <- 2.71828182845905
2 exp = [3 2 3] # exp <- [3 2 3]
3 a = exp(1)     # a <- 3
```

This example is similar to previous example. On the first line, `exp(1)` issues a call to Octave's interpreter for internal function, since `exp` is not defined. Value of `a` is 2.7183. However on the third line, `exp(1)` indexes a matrix that is defined on the second line, which means that the value of `a` is now three.

3.3 Code generation

In scope of this project, two different targets are implemented, Lua and LuaJIT 2.0 bytecode.

3.3.1 Lua

First, Lua target was implemented because of easier debugging and faster development. Octave and Lua have similar scoping rules, which made the implementation fairly easy, though some differences needed to be covered:

- Local functions. In Octave, locally defined functions are visible before their definition. Difference can be viewed on listings 3.8 and 3.9:

Listing 3.8: Scopes of functions in Octave

```
1  function foo()
2      bar()
3  end
4
5  function bar()
6      printf("The winter has come")
7  end
```

In Lua, direct translation would be on listing 3.9.

Listing 3.9: Scopes of local functions in Lua

```
1  local foo = function()
2      bar()
3  end
4
5  local bar = function()
6      print("The winter has come")
7  end
```

In Lua, this does not work. At the time of function foo definition, bar is not defined, which means that the value of bar inside foo function is nil. A static register, holding a map of functions was created, to handle this problem. Resulting code on listing 3.10

Listing 3.10: Local function table fixes function scoping in Lua

```
1  local functions = {}
2  functions["foo"] = function()
3      functions["bar"]()
4  end
5
6  functions["bar"] = function()
7      print("The winter has come")
8  end
```

- In Octave, iterator in `for` loop is visible in outer scope. Example for Octave behaviour is on listing 3.11.

Listing 3.11: Iterator scope in Octave

```
1  a = 1
2  for a = [2, 3]
3      continue
4  end
5  assert(a == 3)
```

A direct translation to Lua would result to a code on listing 3.12.

Listing 3.12: Iterator scope in Lua

```
1  local a = 1
2  for _, a in ipairs({2, 3}) do end
3  assert(a == 3)
```

Here, a's value in assert statement is one and we get a bad assertion. This was fixed with the introducion of a new local variable `_iter_`. An example can be seen on listing 3.13.

Listing 3.13: Special iterator variable in Lua to conform to Octave's scoping rules

```
1  local a = 1
2  for _, _iter_ in ipairs({2, 3}) do
3      a = _iter_
4  end
5  assert(a == 3)
```

- Continue statement is missing in Lua 5.1. Fortunately, LuaJIT 2.0 implements feature from Lua 5.2 called *goto*. This can be and is used for emulating continue statement. See listing 3.14

Listing 3.14: continue statement in LuaJIT 2.0

```
1  for _, a in ipairs({2, 3}) do
2      if a < 3 then
```

```
3                 goto continue
4             end
5             print("this is printed only once")
6             ::continue::
7         end
```

3.3.2 LuaJIT 2.0 bytecode

LuaJIT 2.0 bytecode is defined in section 2.2. For easier development we used two community projects:

- Luajit lang toolkit [37], which includes Lua to LuaJIT 2.0 bytecode compiler that annotates the generated code with additional information.
- Brozula [38] is a LuaJIT 2.0 Bytecode compiler that generates ES5 javascript code. It was used mainly because it generates much more debuggable errors in case of broken LuaJIT 2.0 bytecode files. When LuaJIT 2.0 recieves a broken bytecode file, it just exits with a bad file format error.

Even with the use of these tools, classical debugging tools were still missing. That is what lead us to implement some additional debugging features:

- Output of annotated, ASCII encoded bytecode. With this method, we ware able to compare the our output to the output of LuaJIT lang toolkit.
- Instruction injection. Injecting additional instructions in between generated instructions enabled us to output the state of registers for every executed instruction, or to just track execution of instructions.

These debugging tools are still available as command line arguments, which are `--luax-log-instructions`, `--luax-log-registers` for following

the instructions and registers at runtime and `--show-generated-code` to show the code generated by the compiler.

The struggles of implementing this target are documented in sections 4.7. Comparing to implementing Lua target, bytecode was harder to implement because of lacking documentation. Problems that were present with Lua output (see section 3.3.1) are not a problem in generating bytecode, since we have much more control over the scope of registers.

3.4 Runtime system

In order to be able to execute Octave code, LuaJIT 2.0 VM needs to have access to Octave's *liboctiterp* library, so two libraries were written in order to be able to access functionalities, implemented in *liboctinterp*. Runtime system refers to these libraries and the program, that actually links all the libraries into a workable system, which controls the execution.

3.4.1 Rust to Octave

After input Octave program is read, parsed and compiled to a given target language, a temporary file is created and the compiled program is dumped into that file. Then Rust program initializes Octave's `octave::interpreter` and LuaJIT's `lua_State`, sets all paths and variables and then the program can be run using `luaL_loadfile` function. In case of failure during execution, all exceptions are caught and presented as errors. At the end of the execution, a cleanup is performed to free any used resources.

3.4.2 Lua to Octave

To bind these two languages, we need two libraries, one written in C++ and one written in Lua. C++ is compiled to shared object and linked to LuaJIT 2.0 at runtime. The exposed functionalities from `libinterp` can be divided into structures and functions.

There are two exposed structures,

- `octave_value` represents any Octave value that is supported in this work and listed in section 3.1.5 and
- `octave_value_list` represents function input and output values.

Exposed functions are

- contructors for creating Octave values from Lua values: integer, double and string,
- range constructor, used by ':' range operator,
- arithmetic operations for unary (eg. a++) and binary (eg. $a + b$) oprations and special assignments (eg. a += 1),
- conversion of any `octave_value` to boolean,
- matrix properties: ndims, columns, rows, size and length,
- indexing matrix,
- calling internal Octave functions,
- `octave_value_list` constructor, accessor and destructor and
- converting any `octave_value` to string for displaying purposes.

3.5 Work division between LuaJIT and Octave

When compiled code is run in LuaJIT VM, some operations are still performed by Octave's `libinterp` library. This section explains how is the execution of Octave programs divided between the two.

Interpreting an Octave expression can be divided into a couple of steps. An example expression $a = b + 1$ is executed in five steps:

1. parsing $\rightarrow assign(a, sum(b, 1))$,
2. fetch parameters as `octave_value`s for b and for *1*,
3. determine sum function based of types of b and *1*,
4. execute function sum,
5. store result in a variable a.

After the code is compiled to Lua, then parsing and fetching of parameters are performed by LuaJIT 2.0. The next two steps are executed within `libinterp` library and storing is performed by LuaJIT 2.0

Garbage collection is done by LuaJIT 2.0, but reference counting is still implemented as part of `octave_value` type in *liboctinterp* library. This reference counting presents an unnecessary overhead that was not removed, since that would demand forking the Octave project.

In next two section we present work division in other structures of Octave programs.

3.5.1 Loops and conditionals

Octave implements three kinds of loops, `for`, `do until` and `while`. These structures are all implemented with LuaJIT control structures, only two operations are performed within Octave:

- conditional statement in `while`, `do until` and `if` blocks are computed inside Octave,
- to handle `for` loop, a table is created holding value for every iteration. Iteration is then executed using ipairs function in Lua standard library.

3.5.2 Functions

Functions are also implemented in Lua, but there is a question of multiple return and variable number of input parameters. These features are implemented using Lua tables. This also means that locally defined functions must

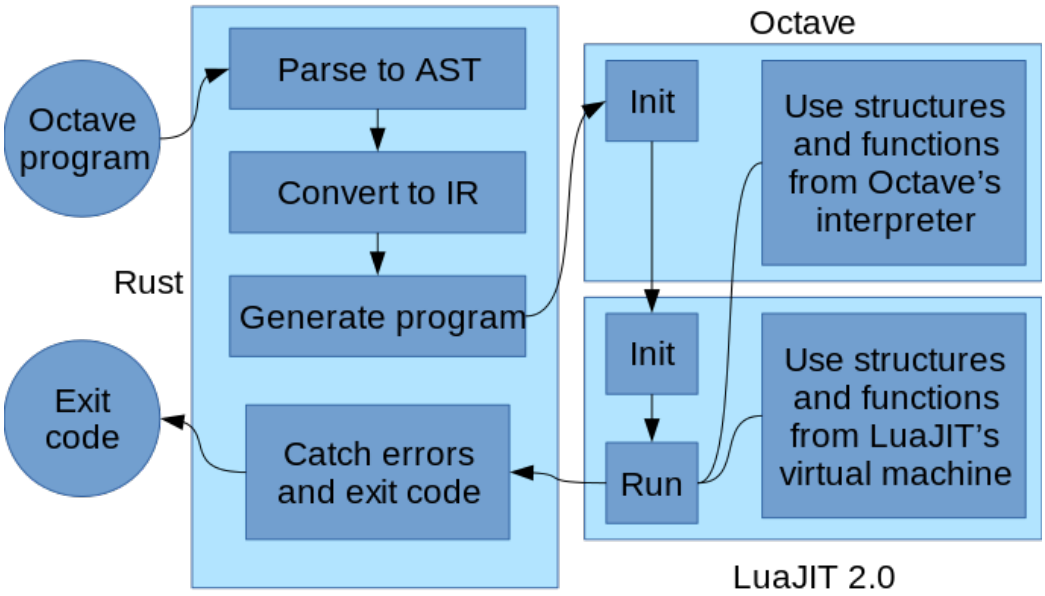


Figure 3.2: Overview of compiler and runtime control flow when running an Octave program

be called differently than internal Octave functions, but that is all handled transparently for the user.

3.5.3 Control flow

For a better visual representation of control flow and work division in our compiler and runtime, a simple diagram was created to illustrate the sequence of actions performed when running an Octave program. It is shown on figure 3.2.

Chapter 4

Development process

We started the development with implementing all Octave structures within Lua, where metatables would be used fo dynamic typing. Soon we realized that amount of work needed to implement a subset of types from *liboctave* would be way too high, so we instead focused on supporting `octave_value` as a type for all Octave values and dynamic typing already implemented with this type was used.

4.1 Grammar

At the time when a choice of grammar parser generation library had to be made, rust-peg was the leading such library for Rust. For every implemented structure in grammar, a test was written to check if a new structure is being parsed correctly. These tests can still be found within the source code, in file grammar/test.rs.

For parsing expressions the grammar produced a slow code and the decision was made to change the parsing method. We removed the priority of operators from the grammar and parsed expression as a flat sequence of operations. A shunting yard algorithm first described by E.W. Dijkstra [39] was written to handle the problem of priorities of different operators. Soon thereafter, rust-peg library added a memoisation functionality, which was

fast enough that we replaced the shunting yard algorithm with original expression parsing grammar. This implementation is still used for determining operator priority in the current state of the project.

4.2 Code generation, first attempt

After finishing writing the grammar, first attempts of code generation into LuaJIT 2.0 bytecode were made. Later, the Octave development progressed and introduced changes to `libinterp` library and some refactoring was needed to make the code compile with upstream changes. The build system changed from premake build configuration [40] to Rust's own Cargo. Bindings from Rust to Lua and Octave were also written to support a new runtime environment, which was rewritten from C++ to Rust.

4.3 LuaJIT 2.0 bindings

Bindings between LuaJIT 2.0 and `liboctinterp` are needed, to run functionality written in `liboctinterp` inside LuaJIT VM. Writing these bindings was test driven, so tests were written for every new functionality added. These tests can still be found in *test/lua* directory and they cover:

- make `octave_value` from Lua value, for strings, integers, doubles etc.,
- `do_binary_op` for any binary operation,
- `do_unary_op`,
- matrix initialization using horzcat and vertcat and
- matrix indexing and modification.

These bindings can be found in folder *src/bindings*, files *octave.lua* and *octave2lua.cpp*.

4.4 Code generation, second attempt

I started creating tests for code generation to Lua, which are located in test/octave directory. After that, a simple one to one transform was implemented that generated output directly from the Octave code, without the use of temporary values in produced Lua code. At this stage, no intermediate representation was used.

This simple approach did not cover some Octave functionality. One example is a chained function call, for example an assignment

```
1 X = A(B)(C)(1:end)
```

can not be transformed directly. In order to determine how big the range in rightmost square bracket is, the length of `A(B)(C)` is needed. However we can't obtain the length of this temporary value, since it is not stored, as can be seen in this line of target Lua code:

```
1 X = bracket_op(bracket_op(bracket_op(A, B), C), range(1,
    len(?)))
```

As we can see, a temporary value is needed to store the intermediate value. These temporary values were introduced in a new code generation system that would produce Lua code as seen here:

```
1 tmp1 = bracket_op(A, B)
2 tmp2 = bracket_op(tmp1, C)
3 tmp3 = range(1, length(tmp2))
4 tmp4 = bracket_op(tmp2, tmp3)
5 X = tmp4
```

To use this system, a simple templating language was created. Templating language was constructed from a string and a following vector of templates. A vector was used to fill the templating tags within the string. Two kinds of tags were defined:

- `[:i]` takes i-th element from the vector and

- `[Dx]` takes the rest of elements from the vector and joins it using x as a delimiter.

The code from listing 4.4 would be constructed from a template presented on listing 4.1.

Listing 4.1: Example code template for Lua target

```
Template("[:0]\\nX=[:1]", [
 Template("[D\\n]", [
  Template("[:0] = [:1]", [
   Template("tmp1", []), Template("do_bracket_op([:0],
       [:1])", [
    Template("A", []), Template("B", [])
   ])
  ]),
  Template("[:0] = [:1]", [
   Template("tmp2", []), Template("do_bracket_op([:0],
       [:1])", [
    Template("tmp1", []), Template("C", [])
   ])
  ]),
  Template("[:0] = [:1]", [
   Template("tmp3", []), Template("range([:0], [:1])", [
    Template("1", [])
    Template("length([0])", [Template("tmp2", [])])
   ])
  ]),
  Template("[:0] = [:1]", [
   Template("tmp4", []), Template("do_bracket_op([:0],
       [:1])", [
    Template("tmp2", []), Template("tmp3", [])
   ])
  ])
 ]),
 Template("tmp4", [])
])
```

For code generation, a structure is used to keep the state of compilation. It contains multiple fields:

`function_scope` is a boolean that is set if currently parsed part of AST is inside a function. It is used to choose between the two sets of local and global variables, depending on a scope.

`function_locals` **and** `main_locals` represent all defined local variables in a current scope.

`function_globals` **and** `main_globals` represent all defined global variables in a current scope.

`max_locals` is a scalar, which equals the number of local variables used in current scope. It is useful for efficient use of registers, determining frame size in LuaJIT 2.0 bytecode target and checking if maximal limit of registers is reached.

`disable_identificator_mangling` is a boolean that enables or disables identificator mangling. This is the process of assigning register numbers to variable names. Disabling it is used for parsing one word commands. One example would be a command *true*. If true is not locally or globally defined variable, than it can only be a call to Octave's integrated function true and as such, a name has to be preserved and not changed to a register number.

`callers` is a stack of variables that are the left hand side of bracket call. Example: A(B(C))(D) - first A is pushed on the stack, then B is pushed on the stack, then both are popped. Here, A(B(C)) is assigned to temporary variable `tmp` and this variable is pushed and pulled from the stack. This structure is needed, when argument value depends on caller value. In given example, if D was `1:end`, then the value of `end` needs to be calculated from a value on top of callers stack, which in this case is `tmp`.

`operator_num` is the index of the argument in arguments for a function call or matrix indexing. In example: A(1:end, 2:end), the `operator_num` is needed to determine the value of "end".

4.5 Garbage collection

Up to this point, no garbage collection was done by LuaJIT 2.0 VM. This created memory leaks and slowed the execution with cache misses. A function called gc inside LuaJIT's ffi library was used to garbage collect the `octave_value`s that were created as part of the program execution.

Contrary to `octave_value` type, `octave_value_list` does not need to be tracked by a garbage collector, instead it can be manually deleted because its scope is known at compile time. This removed all memory leaks and other memory related bugs.

4.6 Optimizations

At this point, first benchmark was created to test the speed of this compiler. Some possible performance improvements were found that the we were able to optimize.

4.6.1 Function caching

Octave uses `std::map` for its internal map of function names to functions themselves. Time complexity of accessing values in a `std::map` is logarithmic, as defined in C++ standard [41], page 717. Functions are cached in a Lua table that is implemented as a hash table in LuaJIT 2.0. Accessing values by a key takes amortized constant average time. This achieved on average of 2% improvement on benchmarks from section 5.2.

4.6.2 Make list implementation

Constructor for `octave_value_list` that is used for input and output parameter for all Octave functions, does not allocate any space for Octave values by default. So a sequence of instructions on listing 4.2 reallocates the array three times. This was easily fixed by pre-allocating the space to correct size. This brought on average of 3% improvement benchmarks from appendix 5.2.

Listing 4.2: Creating octave_value_list, naively reallocating

```
1 -- x, y and z are octave_values
2 A = make_octave_list()
3 set_nth(A, 0, x)
4 set_nth(A, 1, y)
5 set_nth(A, 2, z)
```

4.6.3 Memory pool

Every `octave_value` and `octave_value_list` in Lua is allocated on the head and is used as raw pointers, which means that space is allocated for every value, even for shortly lived temporaries. Allocating many values of identical size on heap is a problem that can be improved with the use of a special memory pool instead of system allocator. The difference between a system allocator and special memory pool allocator ranges from 2% to 5%. Speeds for three benchmarks can be seen on figure 4.1. Memory pool implementation that is used is open source and available online [42].

4.7 Intermediate representation

At this point in time, all functionality but LuaJIT 2.0 bytecode target language was written. The problem with the current implementation of Lua target was that is simply was not suited for outputting the bytecode. This problem was solved with the introduction of intermediate representation,

which was designed in such a way that supported both targets, Lua and LuaJIT 2.0 bytecode. IR's structure is explained in section 3.2.

Labels

Control structures, these are loops and conditional if statements, are implemented using conditional jumps in LuaJIT 2.0 bytecode target. To handle these jumps, labels are added to certain instructions. This enables jump instructions (JMP) to be relative to these labels. In the final pass of the compiler, labels are resolved into numeric values, so that jumps can be expressed with JMP. While implementing this feature, we were not able to remove a certain bug that made bytecode fail to execute properly, but only in some strange and rare conditions. The bug was finally found and the root of the bug was author's incorrect knowledge of operator priority in Rust, where the assumed order of execution was different than the actual order. The problematic line of code was:

```
1 label = line_num << 32 + context;
```

4.8 Targets

After completing IR, both targets were implemented. Lua target also contains a code formatter that converts machine generated Lua code into human readable Lua source. This formatter helps with debugging and understanding the compiler output. Target language can be determined via command line argument *-t*, possible choices are lua and luax.

4.9 Comparing Octave programs to *oct files*

Previous work by Jens Rucknagel [6] attempted to specialize dynamic types using type inference and compiling generated code to C++. This provided

large speedups, when types and shapes can be determined at compile time. Simple examples execute up to 40 times faster.

This work shows that there is overhead of Octave's interpreter interpreting Octave programs. To measure the overhead of parsing, *oct* programs (described in section 1) were created for sorting an array using bubble sort and for calculating Fibonacci numbers. This code is a direct translation from Octave programs in listings A.1 and A.3 to two *oct* programs that use `octave_value` as the types of variables.

On figure 4.2 are the measurements of this program. Average speedup achieved was 6.4. We can see that 40 times performance improvement in [6] can in part be attributed to using C++ as target language, without performing additional optimizations. In the next section, we test additional possible speedups using *oct programs*. These speedups were studied for a possible implementation within the Lua and LuaJIT 2.0 bytecode targets, but were not implemented in the final work.

In Octave programs, using constant literals is common. Octave's interpreter allocates new `octave_value` for any such value that could alternatively be defined as a constant value. Using constant static values for these constants, bubble sort program was on average faster by 5% and the Fibonacci program by 15%.

Octave uses std::map to store operator functions. While an expression `b .* c` is executed with Octave's interpreter, a lookup into a key value store is made to locate the actual function, based on types of b and c. As described in 4.6.1, using std::map as a key value store does not give optimal performance. We can measure the time that is used by std::map lookups with function caching, which avoids the lookup. We achieved speedups of 6% for Fibonacci and 3.5% for bubble sort.

The function input and return type in Octave is `octave_value_list`. Internally this class stores `octave_values` in a C++'s std::vector, which causes new heap allocations for every function call and return. To measure the performance impact of these additional operations, we used `octave_value` as

input and output values. However, this does limit us to functions with only one return value. Bubble sort as a function is only called once per execution and most of its execution time is spent within a loop, so there are no speedups with this change. On the other hand, Fibonacci program is highly recursive and the average speedup achieved is 101%, which means that half the time of this program, the time is spent by creating `octave_value_list` type.

These results that can be seen on figures 4.3 and 4.4 give us an estimation of what a possible additional speedup for Octave code is without changing Octave's interpreter or using work from [6] to predetermine the types of variables.

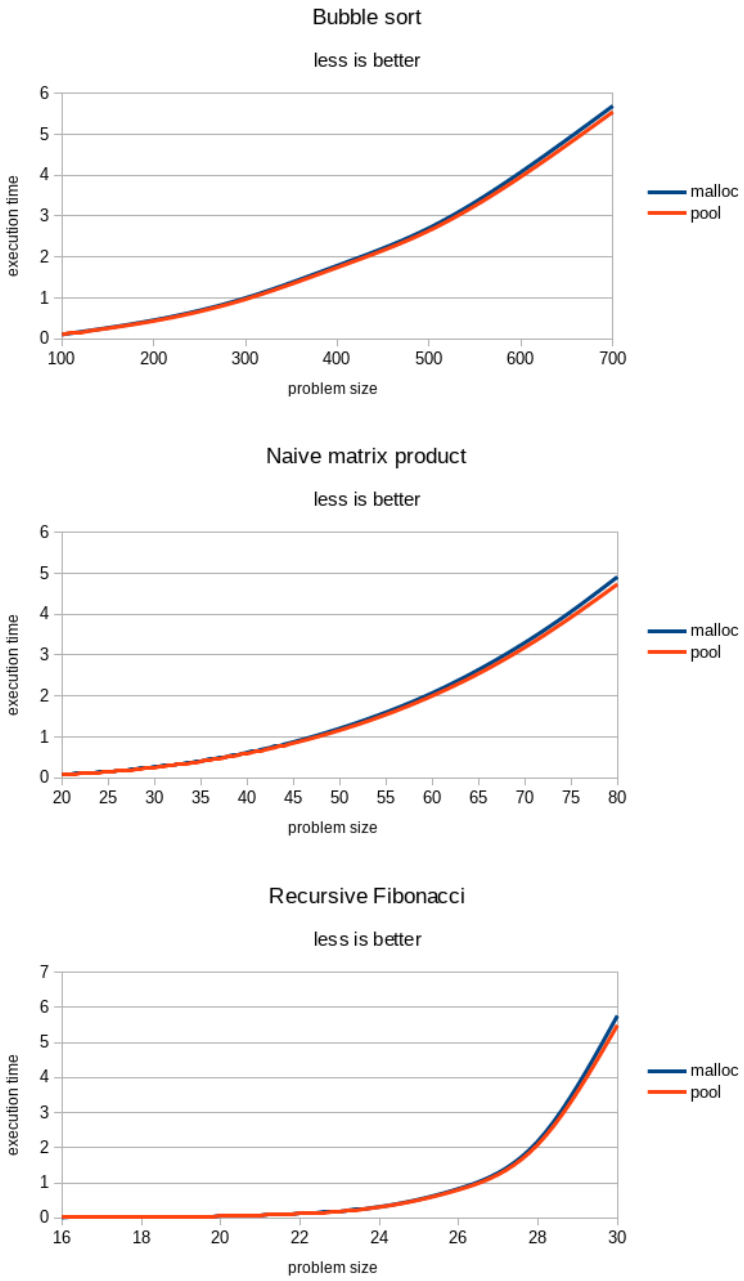


Figure 4.1: Comparison in execution speeds between using custom memory pool allocation versus system malloc allocator.

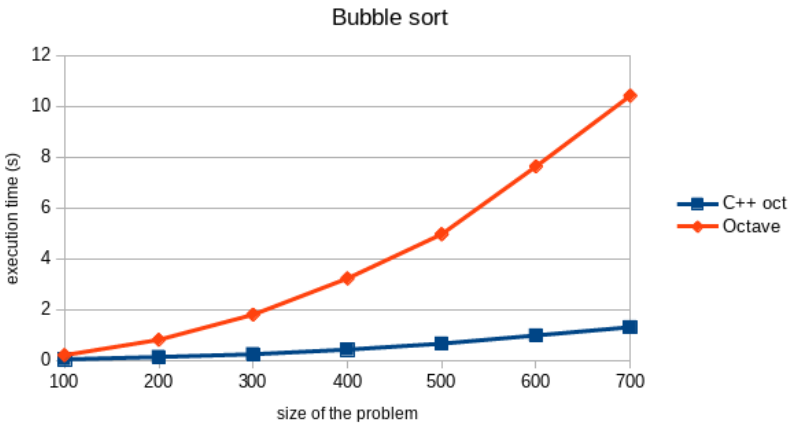


Figure 4.2: Directly comparing the speed of Octave compiled to C++ versus Octave's interpreter for bubble sort programs.

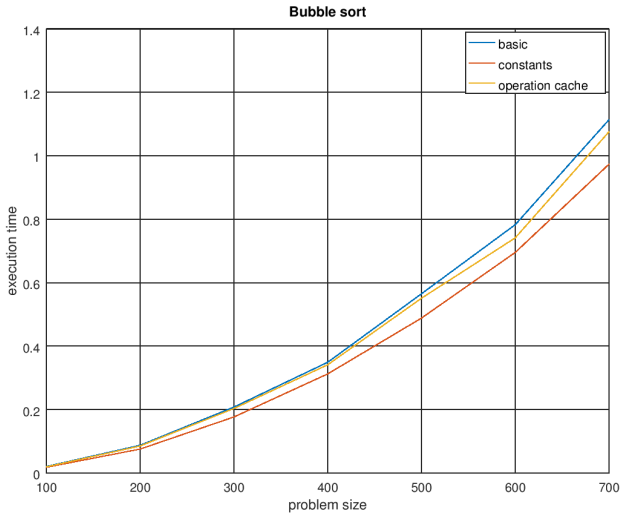


Figure 4.3: Plots of execution speed for bubble sort algorithm implemented as *oct* program using different optimization approaches from section 4.9

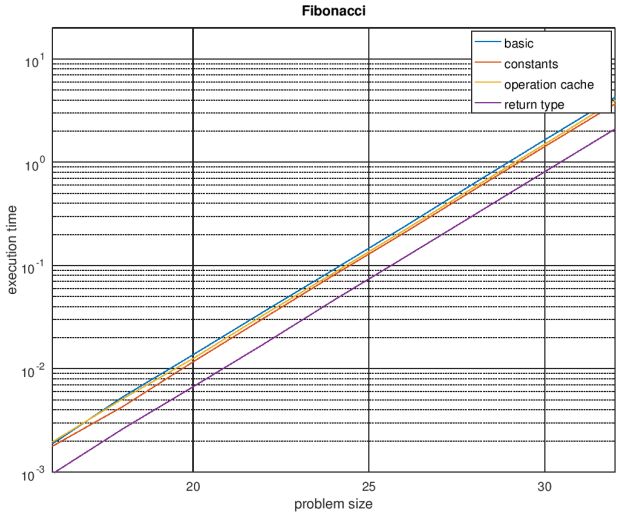


Figure 4.4: Plots of execution speed for recursive fibonacci algorithm implemented as *oct* program using different optimization approaches from section 4.9

Chapter 5

Results

5.1 Testing environment

Every benchmark and measurement in this work was performed on the author's computer. Hardware used: Intel^{R} Core^{TM} i5-3570 CPU @ 3.40GHz with 16 GB of ram. Operating system is Arch distribution of Linux running kernel 4.4. The desktop environment is KDE Plasma version 5. Octave software that was used was a custom build of master branch (revision `23822:9401e88f63cf`) of Octave's repository that can be found on Octave's home page [2].

5.2 Benchmark programs

In order, to measure the speed of Octave to Lua compiler, five benchmarks were written. All tests were run on a computer described in section 5.1

5.2.1 Bubble sort

Bubble sort is a simple $O(n^2)$ sorting algorithm that measures the speed of `while` loop. Actual code can be viewed in appendix A, listing A.1. Testing runs through different lengths of randomly sorted array, from 100 to 700 with steps of 100.

5.2.2 Matrix product

Matrix multiplication can be implemented in multiple forms, most commonly used is Winograd's algorithm [43]. However, we implemented a naive algorithm, which uses three for loops and has worse time complexity as Winograd's algorithm. In our solution, first for loop traverses rows, second iterates over columns and third, twice nested `for` loop, calculates the dot product of a row in first matrix and a column in second. This test measures the speed of `for` loop. Actual code can be viewed in appendix A, listing A.2. Testing runs through multiple sizes of square matrices, beginning with matrix of size 20x20, change the matrix side size by 5 and ending with matrix side of length 70.

5.2.3 Fibonacci

Calculating nth Fibonacci number can be done using a direct formula

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{w\phi - 1},$$

where

$$\phi = \frac{q + \sqrt{5}}{2.}$$

In this work, a recursive implementation with no memoisation is implemented, to measure the speed of a function call. Actual code can be viewed in appendix A, listing A.3. Size of this problem is between 16 and 32 with stepsize of 2.

5.2.4 Eight queens

Eight queens problem is the only benchmark that does not measure a specific feature, but rather overall performance. This algorithm solves this problem:

> How to place eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column,

or diagonal).

A fast algorithm using dynamic programming approach [44] was not used, but rather a brute force search is implemented. This version checks all $8! = 40320$ possible arrangements of queens, if we assume in advance that each queen will be in its own row. This program does not have any input, so only one measurement is performed.

We have implemented two versions of this algorithm, one that is implemented directly and on, where we use two optimions, to decrease execution times. First optimization concerns the use of function `factorial`, which is implemented in Octave using equivalence in equation 5.1.

$$n! = \Gamma(n+1) \tag{5.1}$$

To improve performance, we created our own straight forward implementation of factorial using a for loop. A second not optimal Octave function is `fliplr`, which flips the columns of matrix in the left-right direction (that is, about a vertical axis). This function was rewritten using matrix indexing as in equation 5.2.

$$fliplr(M) = M(:, length(M) : -1 : 1) \tag{5.2}$$

With that we have two implementation of eight queen problem:

- `8queens_v1` uses integrated factorial function and `fliplr`,
- `8queens_v2` uses discussed optimizations.

Version v1 of eight queens program can be viewed in appendix A, listing A.4.

5.3 Target configurations

Every benchmark is executed in five different setups:

Programing language	`Lua target`	`LuaJIT 2.0 bytecode target`
`Bubble sort`	1.89	1.89
`Matrix product`	2.17	2.20
`Fibonacci`	6.27	6.17
`Eight queen, v1`	1.16	1.15
`Eight queen, v2`	1.41	1.40

Table 5.1: Speedups of given target on a given benchmark

- Octave setup uses Octave program directly using Octave's interpreter,
- OML setup executes Lua code with LuaJIT 2.0,
- OMLx setup executes LuaJIT 2.0 bytecode,
- OML nojit setup executes Lua code with LuaJIT 2.0 with JIT turned off and
- OMLx nojit setup executes LuaJIT 2.0 bytecode with JIT turned off.

5.4 Results

Results are presented on figures 5.1, 5.2, 5.3, 5.4 and 5.5. Speedups are calculated using formula 5.3 and listed on table 5.1. Execution times without using JIT techniques in LuaJIT were also measured, speedups are listed on table 5.2

$$S = \frac{t_{old}}{t_{new}} \tag{5.3}$$

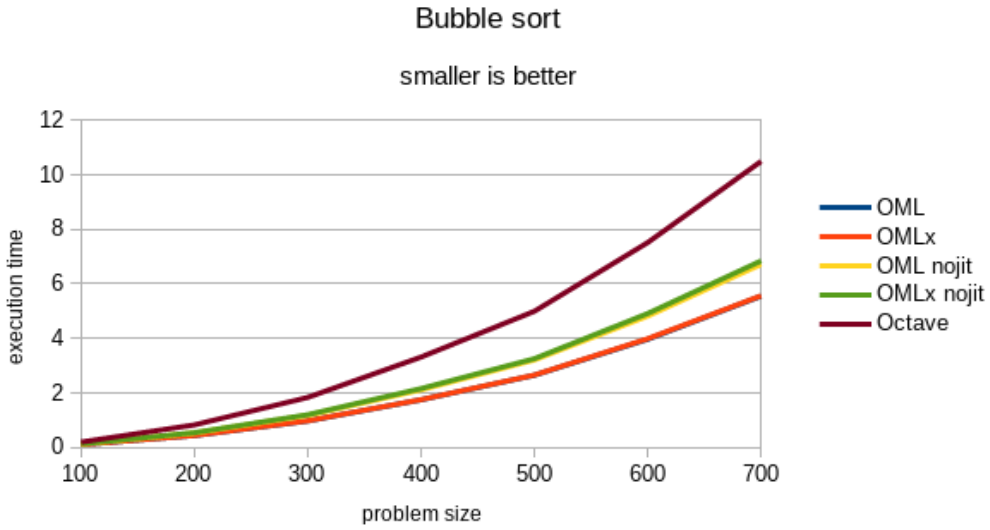


Figure 5.1: Execution speeds for bubble sort as in A.1

Programing language	`Lua target`	`LuaJIT 2.0 bytecode target`
`Bubble sort`	1.56	1.53
`Matrix product`	1.65	1.67
`Fibonacci`	4.53	4.26
`Eight queen, v1`	1.09	1.08
`Eight queen, v2`	1.31	1.29

Table 5.2: Speedups of given target without using JIT on a given benchmark

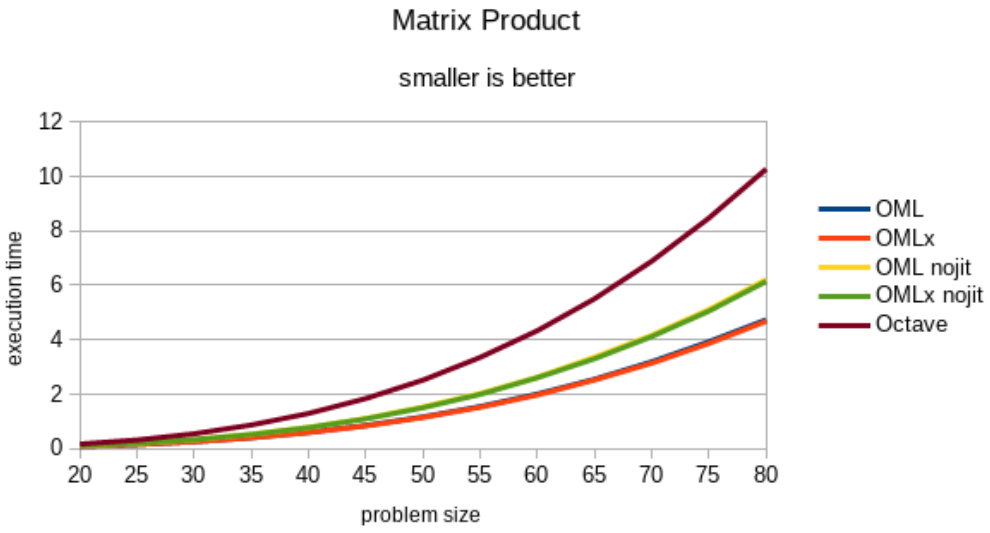


Figure 5.2: Execution speeds for matrix multiplication as in A.2

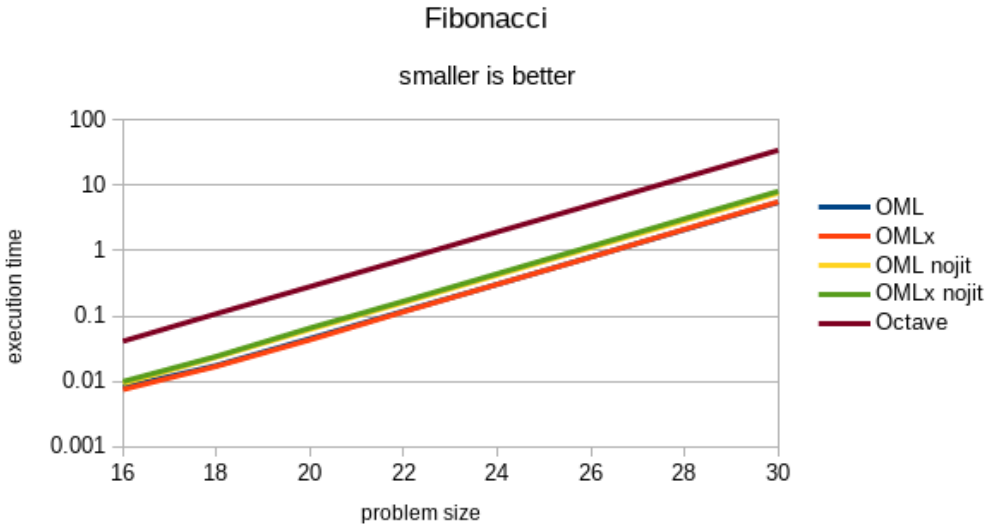


Figure 5.3: Execution speeds for recursive Fibonacci as in A.3

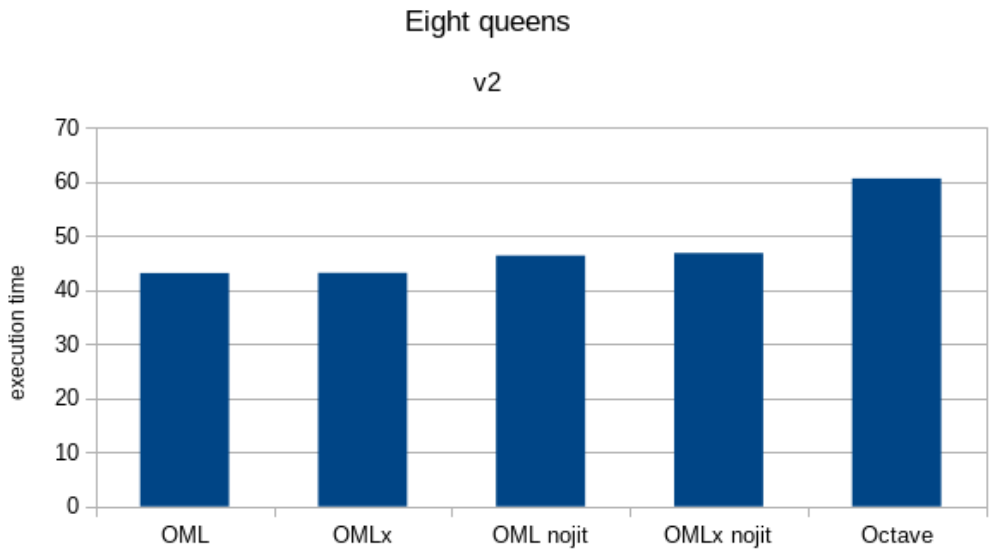


Figure 5.4: Execution speeds for v1 version of Eight queens problem as in A.4

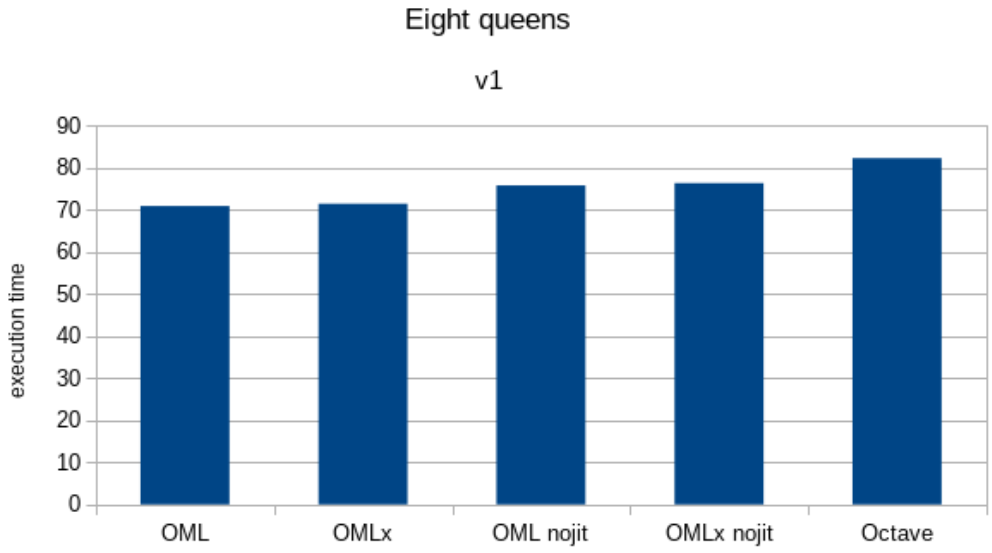


Figure 5.5: Execution speeds for v2 version of Eight queens

Chapter 6

Conclusion

In our work we implemented a compiler we implemented a compiler, that compiles Octave programs to programs for LuaJIT 2.0. The goal of using LuaJIT 2.0 VM was to improve the speed of execution of Octave programs. Two targets are implemented, Lua and LuaJIT 2.0 bytecode and both achieve similar speedups, the second target being on average 1% slower than the first target. Speedups range from 1.1 to 6.27 where smaller speedups are present where most of execution time is inside Octave's integrated functions. Best speedups can be found in recursion heavy problems, where the speed of a function call is significantly faster with LuaJIT VM compared to Octave's interpreter. We also compared execution time of Octave programs compiler to LuaJIT 2.0 with and without JIT compilation. JIT compilation achieves speedups over plain interpretation in range from 1.05 to 1.45. JIT is the most effective on recursion heavy problems.

We also researched the area of possible additional improvements. By replacing Octave's internal structures that represent or deal with Octave values and using our own limited implementation of some of these types, we were able to additionally improve the performance. We also performed profiling on *oct* programs to measure overhead of Octave's internal structures without any time spent in LuaJIT 2.0 VM or Octave's interpreter. The time spent in malloc and free calls for a simple iterative algorithm, that does not

use heap in its C version, was over 30%.

If we were to attempt to improve the speed of executing Octave programs, we would use a different target language, C or C++ and try to improve performance of Octave structures. Dynamic typing is currently implemented via different implementations of class `octave_base_value`, where dynamic casting and virtual method dispatch is used to call methods for different types. Instead a simple jump table is usually implemented and performs better. This combined with decreasing the number of allocations and replacing reference counting with generational garbage collection would provide more performance improvements over only replacing interpreter.

Ah...

Appendix A

Benchmark programs

Listing A.1: Benchmarking program: Bubble sort

```
1  clear
2
3  function ret=bs(vector)
4      len = length(vector);
5      sorted = false;
6      while !sorted
7          sorted = true;
8          i = 1;
9          while i < len
10             if vector(i) > vector(i + 1)
11                 sorted = false;
12                 tmp = vector(i + 1);
13                 vector(i + 1) = vector(i);
14                 vector(i) = tmp;
15             end
16             i += 1;
17         end
18     end
19     ret = vector;
20 end
21
22
23 rand("state", [79, 122, 98, 111, 108, 116]);
```

```
24 for n = 100:100:700
25   start = tic;
26   vec = rand(n, 1);
27   bs(vec);
28   stop = double(tic - start) / 1e6;
29   printf("Bubble sort, %d, %f\n", n, stop);
30   fflush(stdout);
31 end
```

Listing A.2: Benchmarking program: Matrix product

```
1  clear
2
3  function res=mproduct(m1, m2)
4    # only squares are allowed
5    assert(size(m1)(1) == size(m1)(2))
6    assert(size(m1) == size(m2))
7    len = size(m1)(1);
8    res = zeros(size(m1));
9
10   for x = 1:len
11     for y = 1:len
12       for i = 1:len
13         res(x, y) += m1(x, i) * m2(i, y);
14       end
15     end
16   end
17 end
18
19 for n = 20:5:80
20   m = magic(n);
21   start = tic;
22   out = mproduct(m, m);
23   stop = double(tic - start) / 1e6;
24   assert(out == m * m);
25   printf("Matrix triple-loop product, %d, %f\n", n, stop);
26 end
```

Listing A.3: Benchmarking program: Fibonacci

```
clear

function ret = fib(n)
  if n == 1
    ret = 1;
  elseif n == 2
    ret = 1;
  else
    ret = fib(n - 1) + fib(n - 2);
  end
end

for n = 16:2:30
  start = tic;
  fib(n);
  stop = double(tic - start) / 1e6;
  printf("Recursive Fibonacci, %d, %f\n", n, stop);
  fflush(stdout);
end
```

Listing A.4: Benchmarking program: Eight queens

```
function solutions = eight_queens()
    solutions = 0;
    for n = 1:factorial(8)
        nth_permutation(n - 1);
        solutions += check_solution();
    end
    assert(solutions == 92);
end

global matrix;
matrix = [0 0 0 0 0 0 0 0;
          0 0 0 0 0 0 0 0;
          0 0 0 0 0 0 0 0;
          0 0 0 0 0 0 0 0;
          0 0 0 0 0 0 0 0;
```

```
16          0 0 0 0 0 0 0 0;
17          0 0 0 0 0 0 0 0;
18          0 0 0 0 0 0 0 0];
19
20  function pos=nth_permutation(n)
21      global matrix;
22      matrix *= 0;
23      perms_table = [0:7];
24      pos = [];
25
26      for y = 7:-1:0
27          p = factorial(y);
28          i = y + 1;
29          while n - i * p < 0
30              i -= 1;
31          end
32          n -= i * p;
33
34          x = perms_table(i + 1);
35          pos = [pos; x y];
36          matrix(x + 1, y + 1) = 1;
37          perms_table = [perms_table(1:i) perms_table(i+2:end
                )];
38      end
39  end
40
41  function b=check_solution()
42      global matrix;
43
44      m1 = matrix;
45      m2 = fliplr(matrix);
46
47      sums = [
48          sum(diag(m1, -6)),
49          sum(diag(m1, -5)),
50          sum(diag(m1, -4)),
51          sum(diag(m1, -3)),
52          sum(diag(m1, -2)),
```

```
53          sum(diag(m1, -1)),
54          sum(diag(m1, 0)),
55          sum(diag(m1, 1)),
56          sum(diag(m1, 2)),
57          sum(diag(m1, 3)),
58          sum(diag(m1, 4)),
59          sum(diag(m1, 5)),
60          sum(diag(m1, 6)),
61          sum(diag(m2, -6)),
62          sum(diag(m2, -5)),
63          sum(diag(m2, -4)),
64          sum(diag(m2, -3)),
65          sum(diag(m2, -2)),
66          sum(diag(m2, -1)),
67          sum(diag(m2, 0)),
68          sum(diag(m2, 1)),
69          sum(diag(m2, 2)),
70          sum(diag(m2, 3)),
71          sum(diag(m2, 4)),
72          sum(diag(m2, 5)),
73          sum(diag(m2, 6)),
74      ];
75
76      b = sum(sums > 1) == 0;
77  end
78
79  start = tic;
80  eight_queens();
81  stop = double(tic - start) / 1e6;
82  printf("Eight Queens Normal, 0, %f\n", stop);
```

Bibliography

[1] MathWorks, MATLAB, https://www.mathworks.com/products/matlab.html, [Accesed: 1-September-2017].

[2] J. W. Eaton, Gnu Octave, http://www.octave.org, [Accesed: 1-September-2017].

[3] J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J. M. Cardoso, P. C. Diniz, The MATISSE MATLAB compiler, in: Proceedings of the 11th IEEE International Conference Industrial Informatics (INDIN), IEEE, 2013, pp. 602–608.

[4] G. Y. Paulsen, J. Feinberg, X. Cai, B. Nordmoen, H. P. Dahle, Matlab2cpp: A MATLAB-to-C++ code translator, in: Proceedings of the 2016 System of Systems Engineering Conference, IEEE, 2016, pp. 1–5.

[5] P. G. Joisha, P. Banerjee, A translator system for the MATLAB language, Software: Practice and Experience 37 (5) (2007) 535–578.

[6] J. Rücknagel, An Octave type estimator – essential part of an Octave to C++ compiler, Master's thesis, Faculty of Computer Science and Automation at the Technical University of Ilmenau (December 2004).

[7] Lambda the ultimate, online forum, Have tracing JIT compilers won?, http://lambda-the-ultimate.org/node/3851, [Accesed: 1-September-2017] (2010).

[8] M. Pall, Luajit, http://luajit.org/, [Accesed: 1-September-2017].

[9] O. Menegatti, Benchmarks for dynamic languages, https://github.com/gareins/dynamic_benchmarks, [Accesed: 1-September-2017] (2016).

[10] I. Silva, G. B. Moody, An open-source toolbox for analysing and processing physionet databases in matlab and octave, Journal of open research software 2 (1) (2014) 1–4.

[11] N. N. Oosterhof, A. C. Connolly, J. V. Haxby, CoSMoMVPA: multi-modal multivariate pattern analysis of neuroimaging data in MATLAB/GNU Octave, Frontiers in neuroinformatics 10. doi:10.3389/fninf.2016.00027.

[12] F. Milano, An Open Source Power System Analysis Toolbox, IEEE Transactions on Power Systems 20 (3) (2005) 1199–1206.

[13] A. Schlogl, C. Brunner, BioSig: A free and open source software library for BCI research, Computer 41 (10) (2008) 44–50.

[14] R. Crozier, M. Mueller, A new MATLAB and Octave interface to a popular magnetics finite element code, in: Proceedings of 2016 XXII International Conference on Electrical Machines (ICEM), IEEE, 2016, pp. 1251–1257.

[15] A. Garivier, The baum-welch algorithm for hidden markov models: speed comparison between Octave / Python / R / Scilab / MATLAB / C / C++, http://www.math.univ-toulouse.fr/~agarivie/Telecom/code/index.php, [Accesed: 1-September-2017] (2012).

[16] B. K. Rosen, M. N. Wegman, F. K. Zadeck, Global value numbers and redundant computations, in: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 1988, pp. 12–27.

[17] IEEE Standard for Floating-Point Arithmetic, Tech. rep., IEEE (Aug. 2008).

[18] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, D. Padua, FALCON: A MATLAB interactive restructuring compiler, in: Proceedings of 8th International Workshop for Languages and Compilers for Parallel Computing, Berlin, Heidelberg University, 1996, pp. 269–288.

[19] G. Almási, D. Padua, MaJIC: compiling MATLAB for speed and responsiveness, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, 2002, pp. 294–303.

[20] U. Ammann, On code generation in a PASCAL compiler, Software: Practice and Experience 7 (3) (1977) 391–423.

[21] M. W. Ron Petrusha, Luke Latham, Managed Execution Process, in: Getting started with .NET, Microsoft, 2017, pp. 246–249, Accesed: 16-October-2017.

[22] Google Android team, How ART works, `https://source.android.com/devices/tech/dalvik/configure#how_art_works`, [Accesed: 20-September-2017] (Jul. 2017).

[23] J. Aycock, A Brief History of Just-in-time, ACM Computing Surveys 35 (2) (2003) 97–113.

[24] J. G. Mitchell, The design and construction of flexible and efficient interactive programming systems, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (Jun. 1970).

[25] Sun Microsystems, The java hotspot performance engine architecture, `http://www.oracle.com/technetwork/java/whitepaper-135217.html`, Accesed: 20-September-2017 (01 2001).

[26] C. Häubl, H. Mössenböck, Trace-based compilation for the java hotspot virtual machine, in: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, 2011, pp. 129–138.

[27] T. Kistler, M. Franz, Continuous program optimization: A case study, ACM Transactions on Programming Languages and Systems 25 (4) (2003) 500–548.

[28] C. Cascaval, E. Duesterwald, P. F. Sweeney, R. W. Wisniewski, Performance and environment monitoring for continuous program optimization, IBM Journal of Research and Development 50 (2.3) (2006) 239–248.

[29] L. H. d. Figueiredo, R. Ierusalimschy, W. Celes, Lua programming language, `http://lua.org/`, Accesed: 1-September-2017.

[30] I. Gouy, The Computer Language Benchmarks Game, `http://benchmarksgame.alioth.debian.org/`, [Accesed: 1-September-2017] (2017).

[31] K. R. Anderson, D. Rettig, Performing lisp analysis of the fannkuch benchmark, SIGPLAN Lisp Pointers 7 (4) (1994) 2–12.

[32] Wikipedia, Little Endian Base 128, `https://en.wikipedia.org/wiki/LEB128/`, [Accesed: 20-September-2017] (2017).

[33] K. Mehall, Parsing Expression Grammars in Rust, `https://github.com/kevinmehall/rust-peg`, [Accesed: 1-September-2017] (2017).

[34] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Vol. 39, ACM, 2004, pp. 111–122.

[35] M. Richards, C. Whitby-Strevens, BCPL: The Language and Its Compiler, Cambridge University Press, New York, NY, USA, 1981.

[36] D. Dhurjati, S. Kowshik, V. Adve, C. Lattner, Memory safety without runtime checks or garbage collection, in: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, LCTES '03, ACM, New York, NY, USA, 2003, pp. 69–80.

[37] F. Abbate, Luajit lang toolkit, `https://github.com/franko/luajit-lang-toolkit`, [Accesed: 1-September-2017] (2014).

[38] T. Caswell, Brozula, `https://github.com/creationix/brozula`, [Accesed: 1-September-2017] (2012).

[39] E. W. Dijkstra, An ALGOL 60 translator for the X1, Annual Review in Automatic Programming 3 (35) (1961) 329–345.

[40] J. Perkins, Premake build system, `https://premake.github.io/`, Accesed: 20-September-2017 (2017).

[41] Standard for Programming Language *C++*, Standard, International Organization for Standardization, Geneva, CH, ISO 14882:2011(E) (2011).

[42] J. Acay, Memory pool C++ implementation, `https://github.com/cacay/MemoryPool`, [Accesed: 1-September-2017] (2017).

[43] S. Winograd, A new algorithm for inner product, IEEE Transactions on Computers C-17 (7) (1968) 693–694.

[44] I. Rivin, R. Zabih, A dynamic programming solution to the N-queens problem, Information Processing Letters 41 (5) (1992) 253 – 256.